Computer Vision

CSE 455
SVMs and Neural Nets

Linda Shapiro
Professor of Computer Science & Engineering
Professor of Electrical & Computer Engineering

Kernel Machines

- A relatively new learning methodology (1992) derived from statistical learning theory.
- Became famous when it gave accuracy comparable to neural nets in a handwriting recognition class.
- Was introduced to computer vision researchers by Tomaso Poggio at MIT who started using it for face detection and got better results than neural nets.
- Has become very popular and widely used with packages available.

Support Vector Machines (SVM)

- Support vector machines are learning algorithms that try to find a hyperplane that separates the different classes of data the most.
- They are a specific kind of kernel machines based on two key ideas:
  - maximum margin hyperplanes
  - a kernel ‘trick’

The SVM Equation

- $y_{SVM}(x_q) = \underset{c}{\text{argmax}} \sum_{i=1,m} \alpha_{i,c} K(x_i, x_q)$

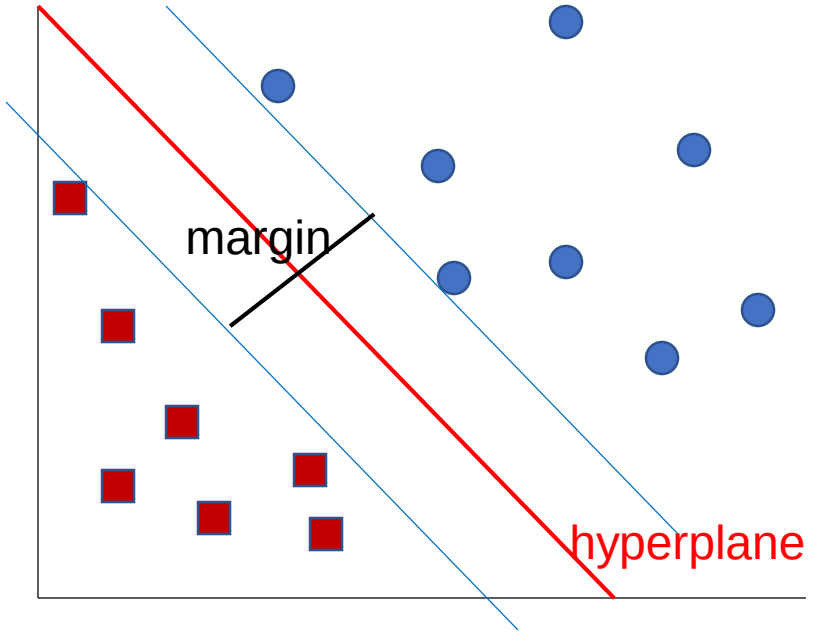
- x_q is a query or unknown object
- c indexes the classes
- there are m support vectors x_i with weights $\alpha_{i,c}$, i=1 to m for class c
- K is the kernel function that compares x_i to x_q

*** This is for multiple class SVMs with support vectors for every class; we'll see a simpler equation for 2 class.

Maximal Margin (2 class problem)

In 2D space,
a hyperplane is
a line.

In 3D space,
it is a plane.



Find the hyperplane with maximal margin for all the points. This originates an optimization problem which has a unique solution.

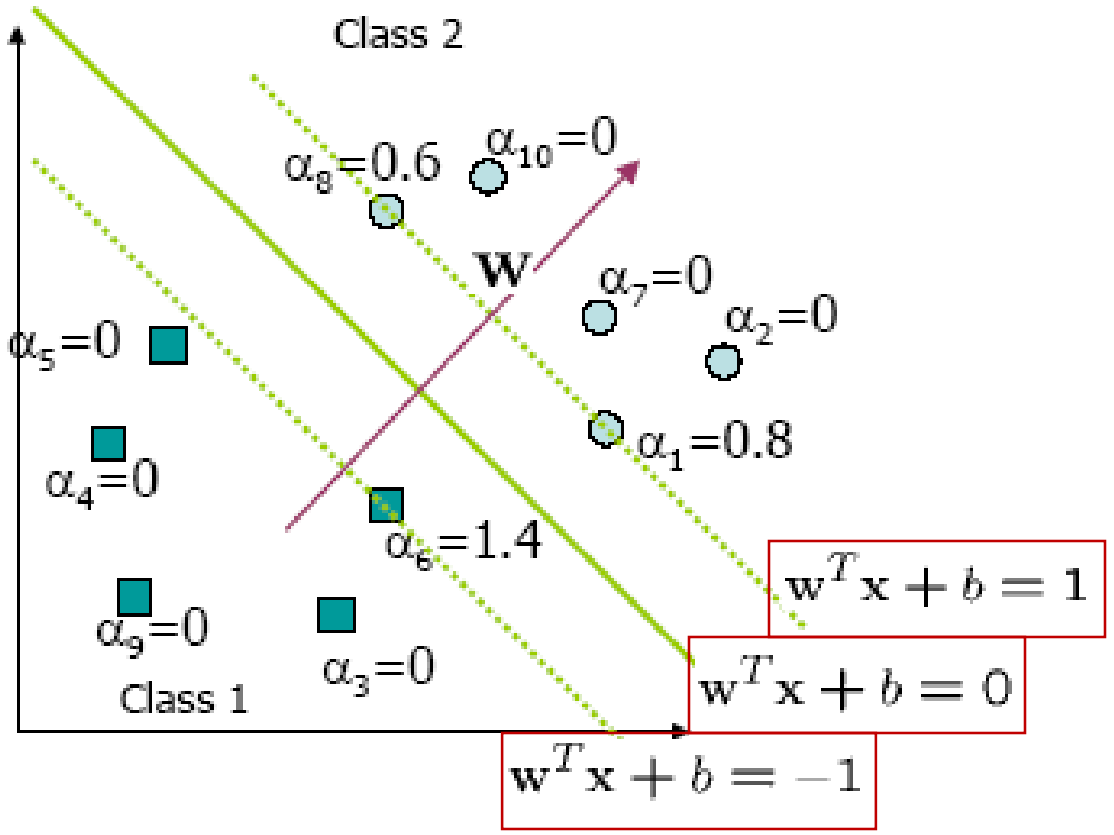
Support Vectors

- The weights α_i associated with data points are zero, except for those points closest to the separator.

- The points with nonzero weights are called the support vectors (because they hold up the separating plane).

- Because there are many fewer support vectors than total data points, the number of parameters defining the optimal separator is small.

A Geometric Interpretation



Kernels

- A kernel is just a similarity function. It takes 2 inputs and decides how similar they are.

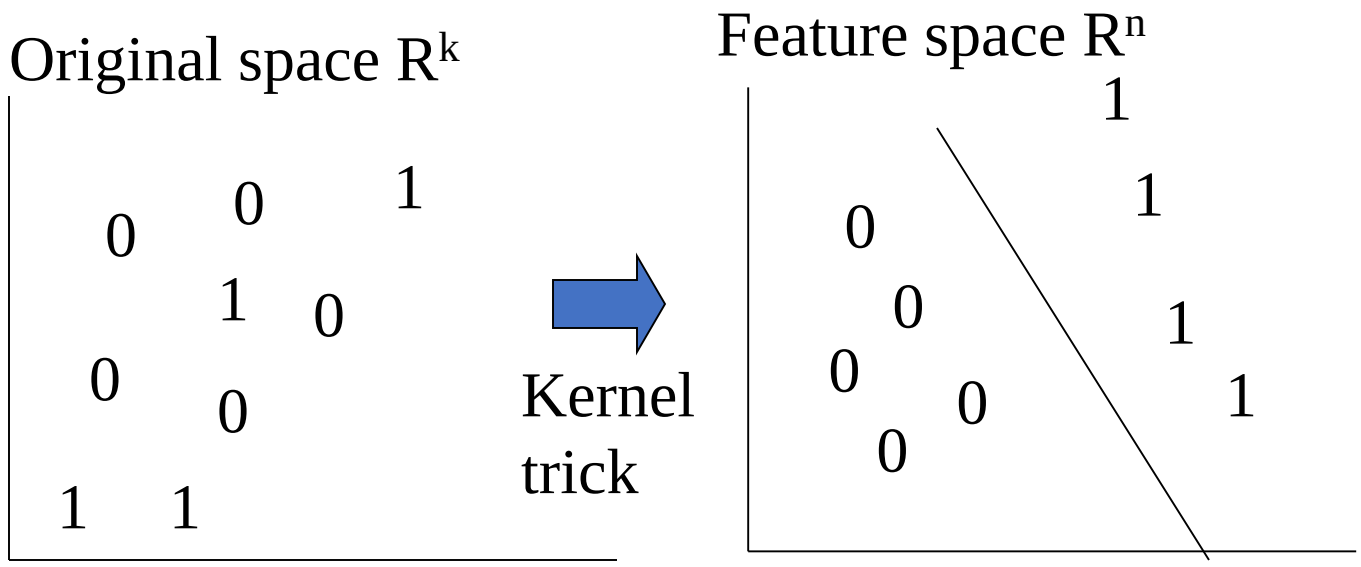
- Kernels offer an alternative to standard feature vectors. Instead of using a bunch of features, you define a single kernel to decide the similarity between two objects.

Kernels and SVMs

- Under some conditions, every kernel function can be expressed as a dot product in a (possibly infinite dimensional) feature space (Mercer's theorem)
- SVM machine learning can be expressed in terms of dot products.
- So SVM machines can use kernels instead of feature vectors.

The Kernel Trick

The SVM algorithm implicitly maps the original data to a feature space of possibly infinite dimension in which data (which is not separable in the original space) becomes separable in the feature space.

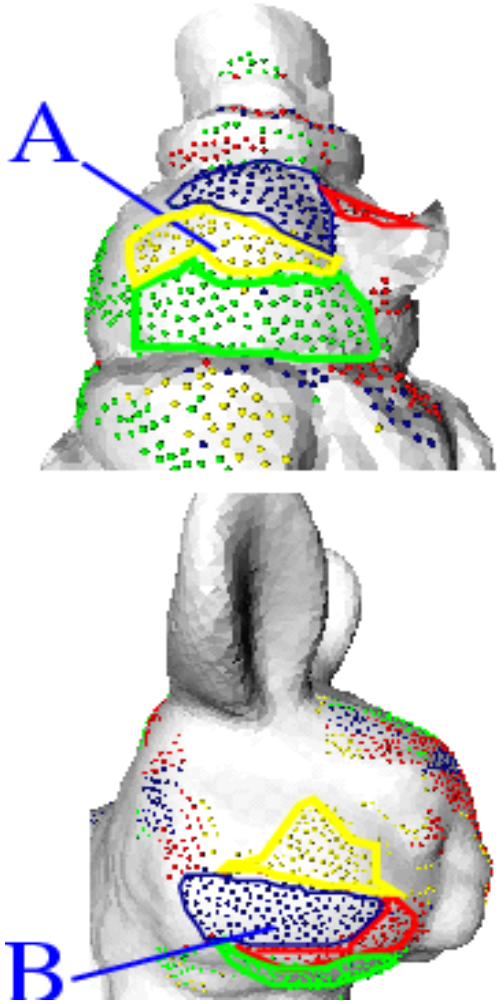


Kernel Functions

- The kernel function is designed by the developer of the SVM.
- It is applied to pairs of input data to evaluate dot products in some corresponding feature space.
- Kernels can be all sorts of functions including polynomials and exponentials.
- Simplest is just the plain dot product: $x_i \bullet x_j$
- The polynomial kernel $K(x_i,x_j) = (x_i \bullet x_j + 1)^p$, where p is a tunable parameter.

Kernel Function used in our 3D Computer Vision Work

- $k(A,B) = \exp(-\theta^2_{AB}/\sigma^2)$
- A and B are shape descriptors (big vectors).
- θ is the angle between these vectors.
- σ^2 is the "width" of the kernel.



What does SVM learning solve?

- The SVM is looking for the best separating plane in its alternate space.
- It solves a quadratic programming optimization problem

$$\underset{\alpha}{argmax} \quad \sum_j \alpha_j \; - 1/2 \sum_{j,k} \alpha_j \, \alpha_k \, y_j \, y_k \, (\mathbf{x}_j \bullet \mathbf{x}_k)$$

subject to $\alpha_j > 0$ and $\Sigma \alpha_j y_j = 0$.

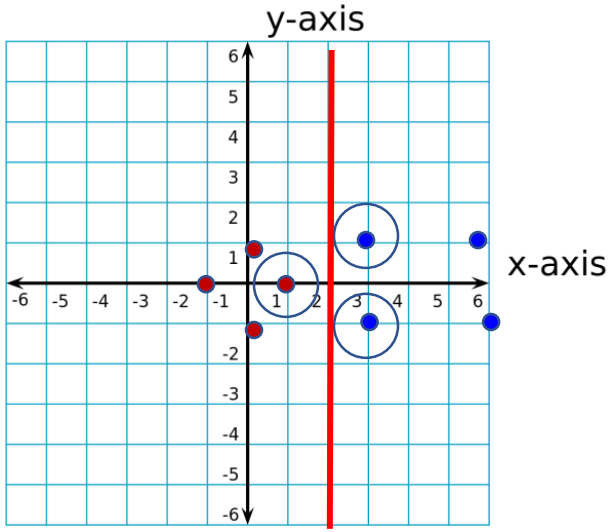
- The equation for the separator for these optimal α_j is

$$h(\mathbf{x}) = \text{sign}(\sum_j \alpha_j \, y_j \, (x \bullet x_j) - b)$$

Simple Example of Classification

- K(A,B) = A • B
- known positive class points {(3,1),(3,-1),(6,1),(6,-1)}
- known negative class points {(1,0),(0,1),(0,-1),(-1,0)}
- support vectors: s = {(1,0),(3,1),(3,-1)} with weights α = -3.5, .75, .75
- classifier equation: $f(x) = \text{sign}(\Sigma_i [\alpha_i * K(s_i,x)] - b)$

b=2



$f(1,1) = \text{sign}(\Sigma_i\ \alpha_i\ s_i \bullet (1,1) - 2)$

= sign(.75*(3,1) • (1,1) + .75*(3,-1)•(1,1)+(-3.5)*(1,0)•(1,1) -2)

= sign(1 – 2) = sign(-1) = - negative class

CORRECT

Time taken to build model: 0.15 seconds

Correctly Classified Instances	319	83.5079 %
Incorrectly Classified Instances	63	16.4921 %
Kappa statistic	0.6685	
Mean absolute error	0.1649	
Root mean squared error	0.4061	
Relative absolute error	33.0372 %	
Root relative squared error	81.1136 %	
Total Number of Instances	382	

TP Rate	FP Rate	Precision	Recall	F-Measure	ROC Area	Class
	0.722	0.056	0.925	0.722	0.811	0.833	cal
	0.944	0.278	0.78	0.944	0.854	0.833	dor
W Avg.	0.835	0.17	0.851	0.835	0.833	0.833	

=== Confusion Matrix ===

a	b	<-- classified as
135	52	a = cal
11	184	b = dor

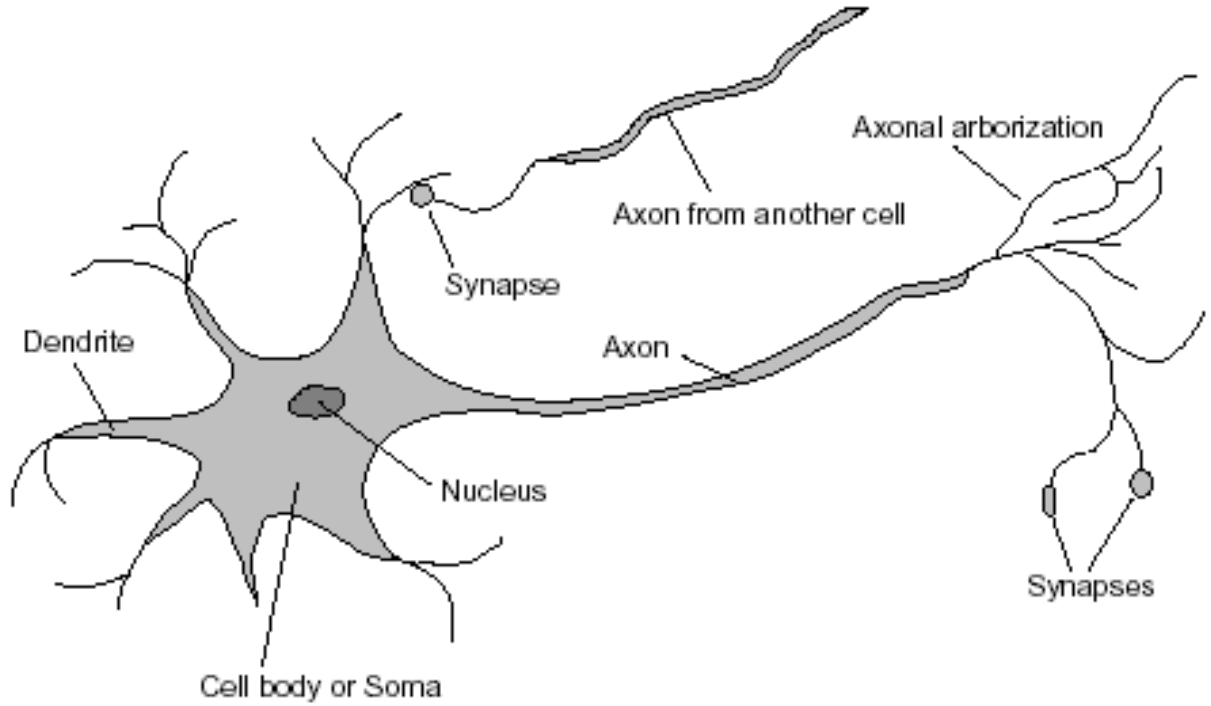
Neural Net Learning

- Motivated by studies of the brain.
- A network of "artificial neurons" that learns a function.
- Doesn't have clear decision rules like decision trees, but highly successful in many different applications. (e.g. face detection)
- We use them frequently in our research.
- I'll be using algorithms from

http://www.cs.mtu.edu/~nilufer/classes/cs4811/2016-spring/lecture-slides/cs4811-neural-net-algorithms.pdf

Brains

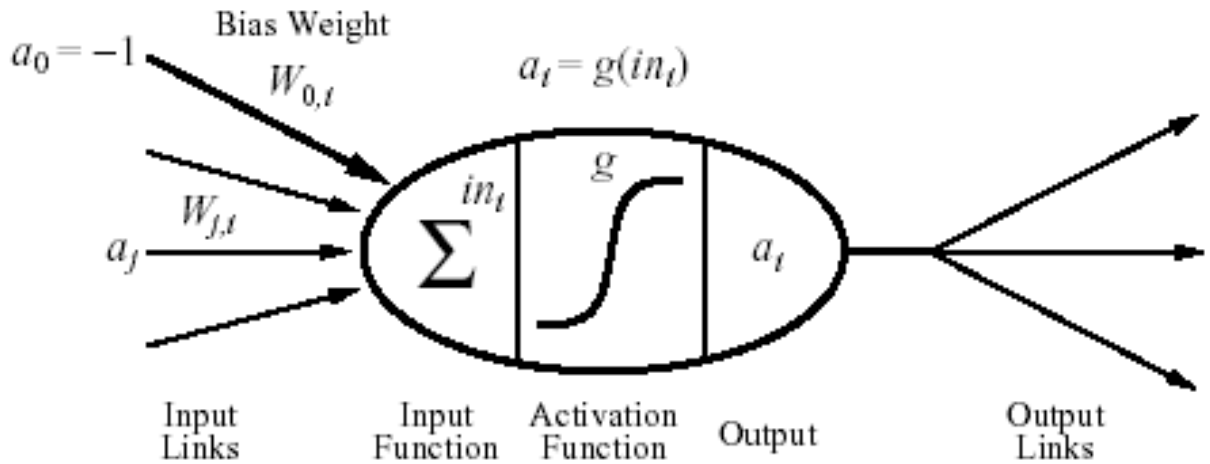
10^{11} neurons of > 20 types, 10^{14} synapses, 1ms–10ms cycle time
Signals are noisy "spike trains" of electrical potential



McCulloch–Pitts "unit"

Output is a "squashed" linear function of the inputs:

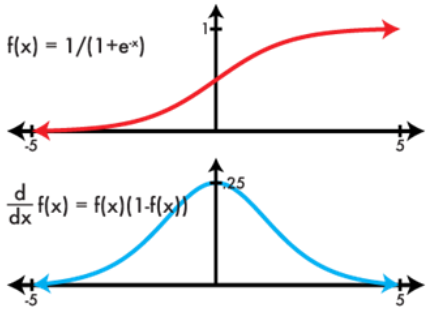
$$a_i \leftarrow g(in_i) = g\left(\sum_j W_{j,i} a_j\right)$$



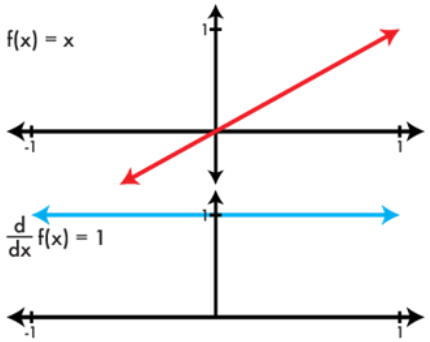
A gross oversimplification of real neurons, but its purpose is to develop understanding of what networks of simple units can do

Common activation functions φ

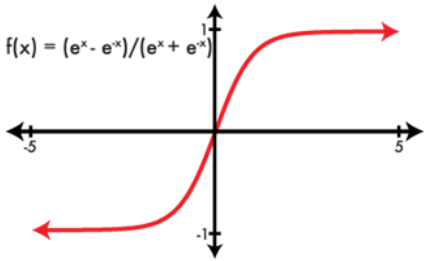
logistic

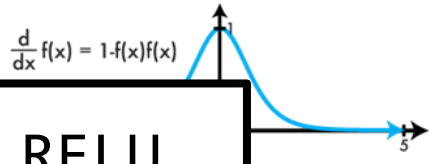


linear

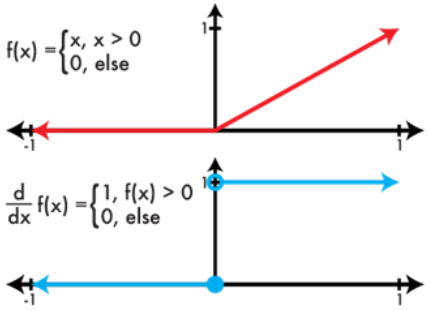


tanh

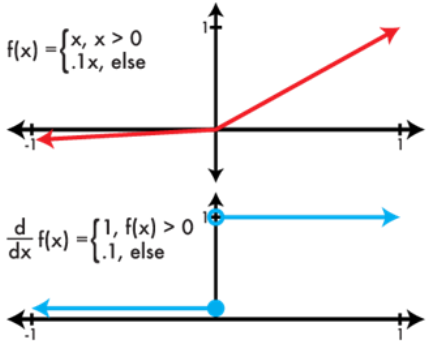




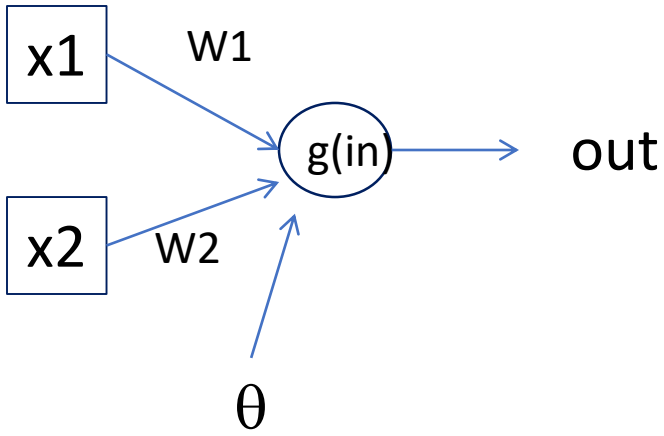
REctified Linear Unit (RELU)



Leaky RELU



Simple Feed-Forward Perceptrons



$in = (\sum W_j x_j) + \theta$

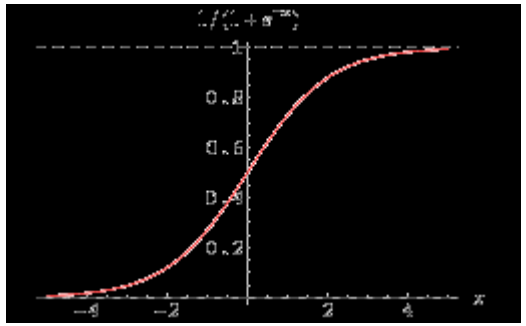
out = g[in]

g is the activation function

It can be a step function:
g(x) = 1 if x >=0 and 0 (or -1) else.

It can be a sigmoid function:
g(x) = 1/(1+exp(-x)).

The sigmoid function is differentiable and can be used in a gradient descent algorithm to update the weights.



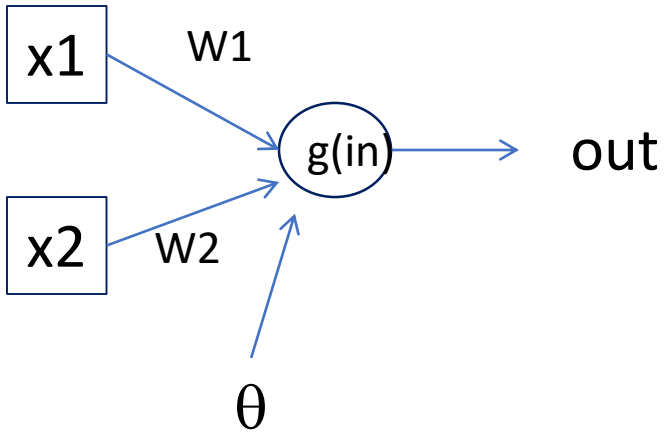
and other things…

Gradient Descent

takes steps proportional to the **negative** of the gradient of a function to find its local minimum

- Let **X** be the inputs, y the class, **W** the weights
- $in = \sum W_j x_j$
- $Err = y - g(in)$
- $E = \frac{1}{2} Err^2$ is the squared error to minimize
- $\partial E/\partial W_j = Err * \partial Err/\partial W_j = Err * \partial/\partial W_j(g(in))(-1)$
- $= -Err * g'(in) * x_j$
- The update is $W_j \leftarrow W_j + \alpha * Err * g'(in) * x_j$
- α is called the learning rate.

Simple Feed-Forward Perceptrons



repeat
  for each e in examples do
    in = $(\sum W_j x_j) + \theta$
    Err = y[e] – g[in]
    $W_j = W_j + \alpha$ Err g'(in) $x_j[e]$
until done

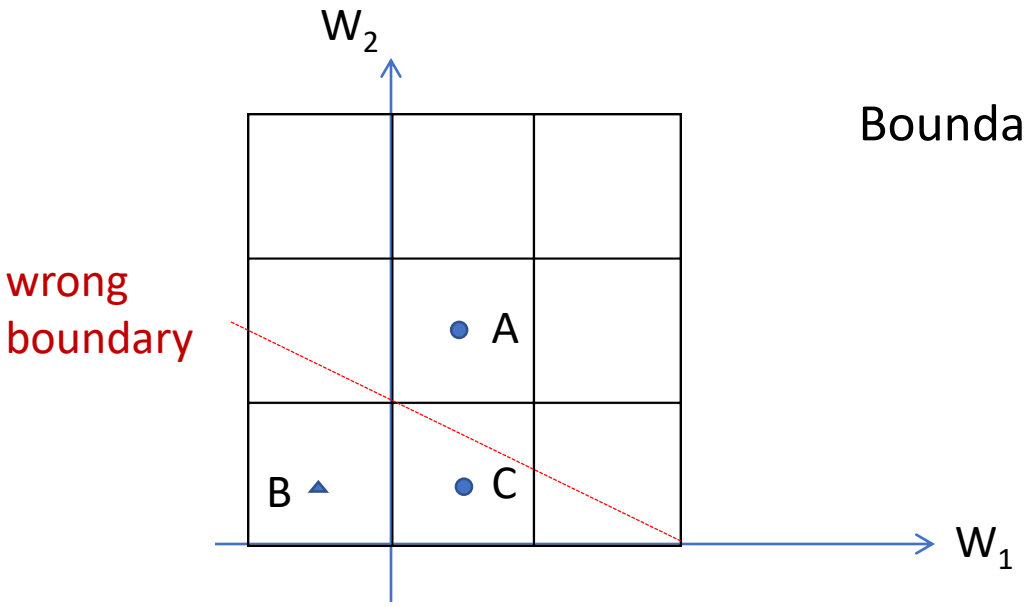
Examples: A=[(.5,1.5),+1], B=[(-.5,.5),-1], C=[(.5,.5),+1]
Initialization: $W_1 = 1$, $W_2 = 2$, $\theta = -2$

Note1: when g is a step function, the g'(in) is removed.
Note2: later in back propagation, Err * g'(in) will be called Δ
We'll let g(x) = 1 if x >=0 else -1

Graphically

Examples: A=[(.5,1.5),+1], B=[(-.5,.5),-1], C=[(.5,.5),+1]

Initialization: $W_1 = 1$, $W_2 = 2$, $\theta = -2$



Boundary is $W_1x_1 + W_2x_2 + \theta = 0$

Learning

repeat
  for each e in examples do
    in = $(\sum W_j x_j) + \theta$
    Err = y[e] – g[in]
    $W_j = W_j + \alpha$ Err g'(in) x_j[e]
until done

Examples:
A=[(.5,1.5),+1],
B=[(-.5,.5),-1],
C=[(.5,.5),+1]
Initialization: $W_1 = 1$, $W_2 = 2$, $\theta = -2$

A=[(.5,1.5),+1]
in = .5(1) + (1.5)(2) -2 = 1.5
g(in) = 1; Err = 0; NO CHANGE

B=[(-.5,.5),-1]
In = (-.5)(1) + (.5)(2) -2 = -1.5
g(in) = -1; Err = 0; NO CHANGE

C=[(.5,.5),+1]
in = (.5)(1) + (.5)(2) – 2 = -.5
g(in) = -1; Err = 1-(-1)=2

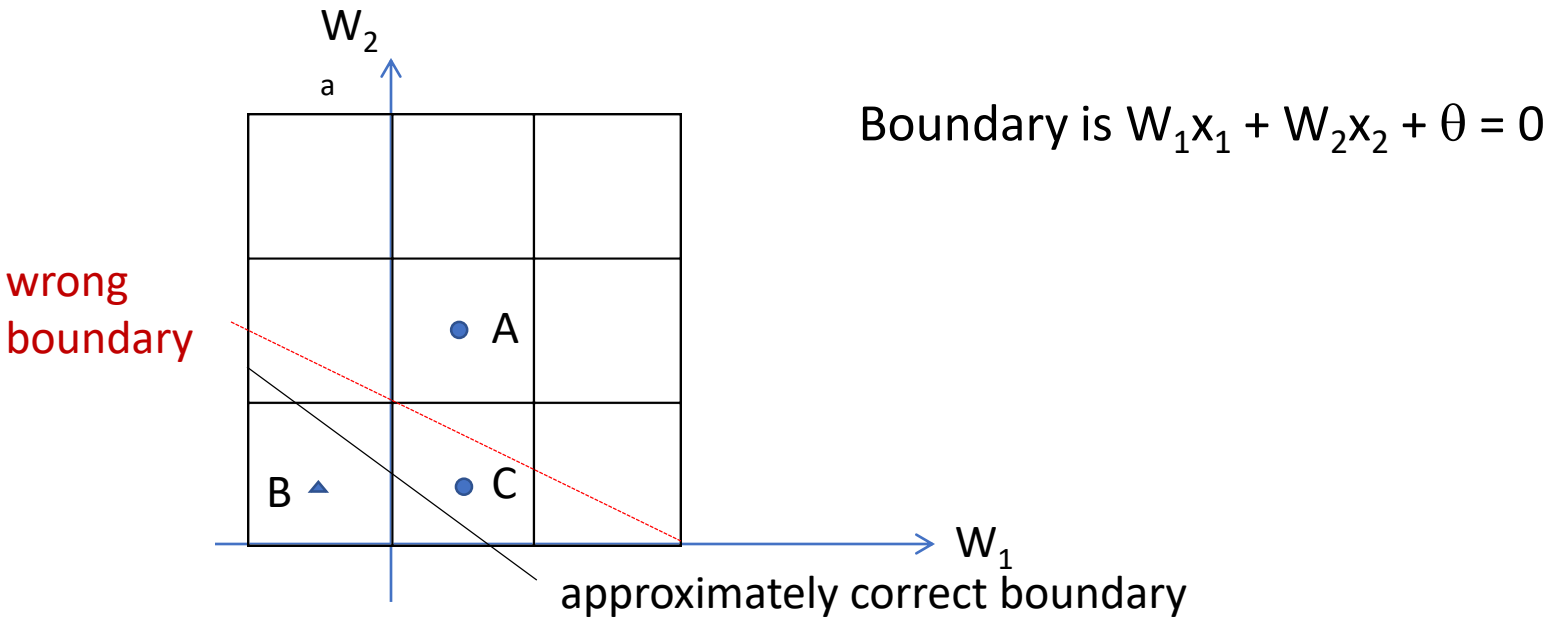
Let α=.5

W1 <- W1 + .5(2) (.5) leaving out g'
  <- 1 + 1(.5) = 1.5
W2 <- W2 + .5(2) (.5)
  <- 2 + 1(.5) = 2.5
θ <- θ + .5(+1 – (-1))
θ <- -2 + .5(2) = -1

Graphically

Examples: A=[(.5,1.5),+1], B=[(-.5,.5),-1], C=[(.5,.5),+1]

Initialization: $W_1 = 1$, $W_2 = 2$, $\theta = -2$



Boundary is $W_1x_1 + W_2x_2 + \theta = 0$

Back Propagation

- Simple single layer networks with feed forward learning were not powerful enough.
- Could only produce simple linear classifiers.
- More powerful networks have multiple hidden layers.
- The learning algorithm is called back propagation, because it computes the error at the end and propagates it back through the weights of the network to the beginning.

The backpropagation algorithm

The following is the backpropagation algorithm for learning in multilayer networks.

function BACK-PROP-LEARNING(*examples, network*)
returns a neural network

inputs:
examples, a set of examples, each with input vector **x** and output vector **y**.
network, a multilayer network with L layers, weights $W_{j,i}$, activation function g
local variables: Δ, a vector of errors, indexed by network node

for each weight $w_{i,j}$ in *network* do
$w_{i,j} \leftarrow$ a small random number
repeat
for each example **(x,y) in** *examples* **do**
/* Propagate the inputs forward to compute the outputs. */
for each node i in the input layer **do** // Simply copy the input values.
$a_i \leftarrow x_i$
for $l = 2$ to L **do** // Feed the values forward.
for each node j in layer l **do**
$in_j \leftarrow \sum_i w_{i,j}\, a_i$
$a_j \leftarrow g(in_j)$
for each node j in the output layer **do** // Compute the error at the output.
$\Delta[j] \leftarrow g'(in_j) \times (y_j - a_j)$
/* Propagate the deltas backward from output layer to input layer */
for $l = L - 1$ **to** 1 **do**
for each node i in layer l **do**
$\Delta[i] \leftarrow g'(in_i) \sum_j w_{i,j}\, \Delta[j]$ // "Blame" a node as much as its weig
/* Update every weight in network using deltas. */
for each weight $w_{i,j}$ in *network* **do**
$w_{i,j} \leftarrow w_{i,j} + \alpha \times a_i \times \Delta[j]$ // Adjust the weights.
until some stopping criterion is satisfied

return *network*

Let's break it into steps.

Initialize

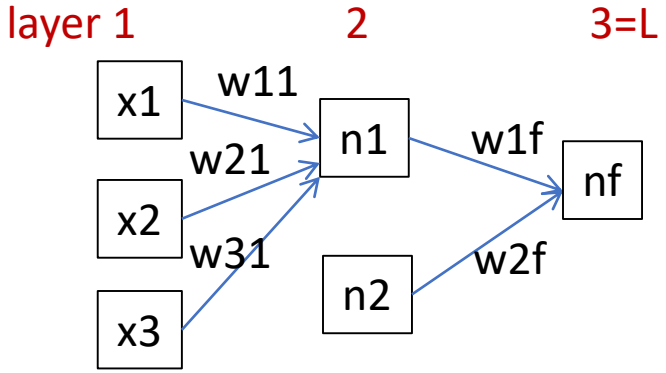
The backpropagation algorithm

The following is the backpropagation algorithm for learning in multilayer networks.

function BACK-PROP-LEARNING(*examples, network*)
returns a neural network

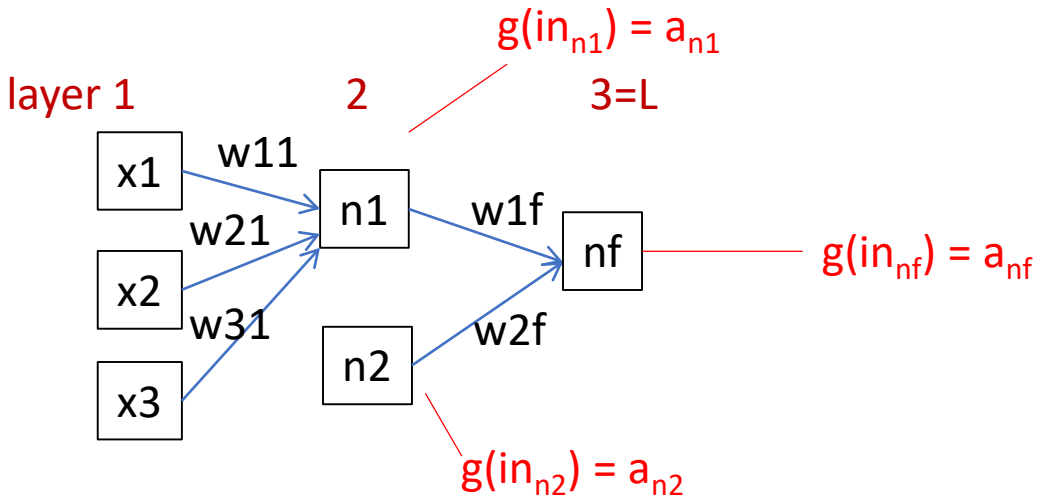
inputs:
examples, a set of examples, each with input vector **x** and output vector **y**.
network, a multilayer network with L layers, weights $W_{j,i}$, activation function g
local variables: Δ, a vector of errors, indexed by network node

for each weight $w_{i,j}$ in *network* do
$w_{i,j} \leftarrow$ a small random number



Forward Computation

repeat
  for each example **(x,y)** **in** *examples* **do**
    /* Propagate the inputs forward to compute the outputs. */
    for each node i in the input layer **do** // Simply copy the input values.
      $a_i \leftarrow x_i$
    for $l = 2$ to L **do** // Feed the values forward.
      for each node j in layer l **do**
        $in_j \leftarrow \sum_i w_{i,j}\ a_i$
        $a_j \leftarrow g(in_j)$

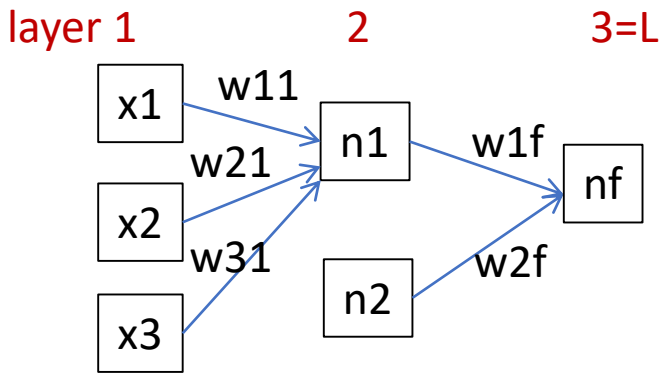


Backward Propagation 1

for each node j in the output layer **do** // Compute the error at the output.

$$\Delta[j] \leftarrow g'(in_j) \times (y_j - a_j)$$

- Node nf is the only node in our output layer.
- Compute the error at that node and multiply by the derivative of the weighted input sum to get the change delta.

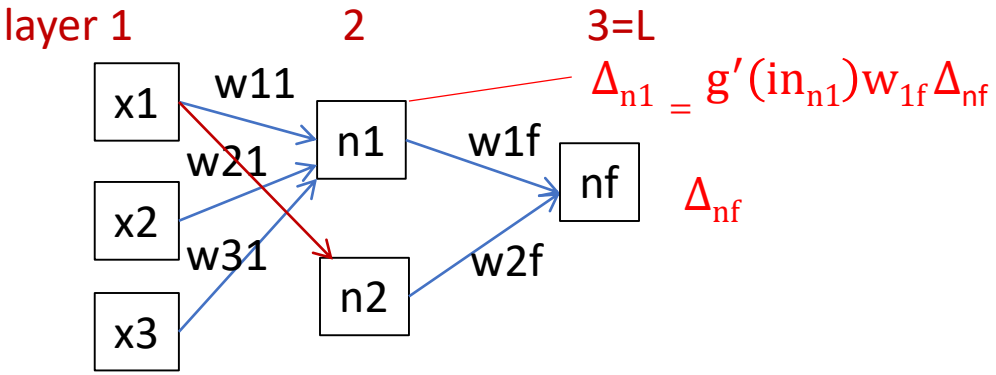


$\Delta_{nf} = g'(in_{nf}) * (y_{nf} - a_{nf})$

Backward Propagation 2

/* Propagate the deltas backward from output layer to input layer */
for $l = L - 1$ **to** 1 **do**
  for each node i in layer l **do**
    $\Delta[i] \leftarrow g'(in_i) \sum_j w_{i,j} \Delta[j]$ // "Blame" a node as much as its weight

- At each of the other layers, the deltas use
  - the derivative of its input sum
  - the sum of its output weights
  - the delta computed for the output error

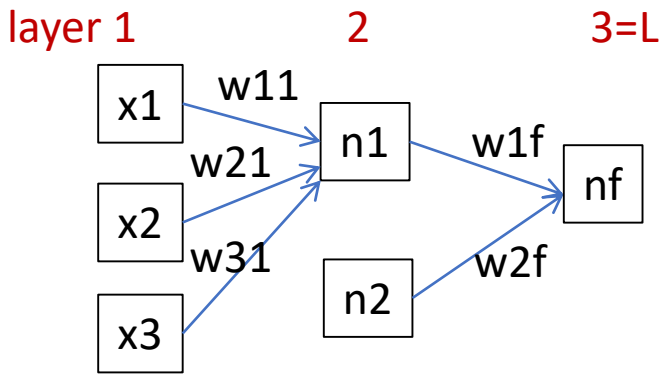


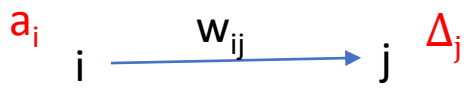
If there were two output nodes, there would be a summation.

Backward Propagation 3

/* Update every weight in network using deltas. */
for each weight $w_{i,j}$ in *network* **do**
$w_{i,j} \leftarrow w_{i,j} + \alpha \times a_i \times \Delta[j]$ // Adjust the weights.

Now that all the deltas are defined, the weight updates just use them.





Back Propagation Summary

- Compute delta values for the output units using observed errors.
- Starting at the output-1 layer
  - repeat
    - propagate delta values back to previous layer
    - till done with all layers
  - update weights for all layers

- This is done for all examples and multiple epochs, till convergence or enough iterations.

Time taken to build model: 16.2 seconds

Correctly Classified Instances	307	80.3665 % (did not boost)
Incorrectly Classified Instances	75	19.6335 %
Kappa statistic	0.6056	
Mean absolute error	0.1982	
Root mean squared error	0.41	
Relative absolute error	39.7113 %	
Root relative squared error	81.9006 %	
Total Number of Instances	382	

	TP Rate	FP Rate	Precision	Recall	F-Measure	ROC Area	Class
	0.706	0.103	0.868	0.706	0.779	0.872	cal
	0.897	0.294	0.761	0.897	0.824	0.872	dor
W Avg.	0.804	0.2	0.814	0.804	0.802	0.872	

=== Confusion Matrix ===

a	b	<-- classified as
132	55	a = cal
20	175	b = dor